In the proceedings of *Formal Methods in Computer Aided Design* (FMCAD), 2007. **Best Paper Award.**

Modeling Time-Triggered Protocols and Verifying Their Real-Time Schedules

Lee Pike
Galois, Inc.
leepike@galois.com

Abstract

Time-triggered systems *are distributed systems in which the nodes are independently-clocked but maintain synchrony with one another. Time-triggered protocols depend on the synchrony assumption the underlying system provides, and the protocols are often formally verified in an untimed or synchronous model based on this assumption. An untimed model is simpler than a real-time model, but it abstracts away timing assumptions that must hold for the model to be valid. In the first part of this paper, we extend previous work by Rushby [1] to prove, using mechanical theorem-proving, that for an arbitrary time-triggered protocol, its real-time implementation satisfies its untimed specification. The second part of this paper shows how the combination of a bounded model-checker and a satisfiability modulo theories (SMT) solver can be used to prove that the timing characteristics of a hardware realization of a protocol satisfy the assumptions of the time-triggered model. The upshot is a formally-verified connection between the untimed specification and the hardware realization of a time-triggered protocol with respect to its timing parameters.*

1 Introduction

Digital control systems are being designed for use in safety-critical contexts such as automobiles ("drive-by-wire") and commercial aircraft ("fly-by-wire") [2, 3]. Safety-critical systems embedded in commercial aircraft must have a failure rate probability no greater than 10^{-9} per hour of operation [4, 5]. A design error causing a system to have a higher rate of failure—say a failure rate of 10^{-8} per hour—is unacceptable, yet it is infeasible to determine whether a system has this reliability through testing alone [6]. The inability to demonstrate correctness through testing motivates us to *prove* these systems are correct.

The specific class of control systems considered in this paper are *time-triggered systems*. Time-triggered systems are implemented as distributed systems in which each node in the system is independently-clocked, and under normal operating conditions, synchronization mechanisms maintain tight synchronization among the local clocks [5]. When the nodes are tightly synchronized, the temporal behavior of the system can be abstracted as if the nodes execute in lock-step. This sort of abstraction is characterized by the *synchronous model* or *untimed model*. At the level of abstraction that the synchronous model provides, formal correctness proofs of the protocols are difficult but feasible [7, 8].

The synchronous abstraction depends on a *realization* (i.e., a concrete implementation—hardware and/or software executing on hardware) satisfying key properties regarding scheduling, message delays, clock skew, message-reception windows, and so forth. A more fine-grained model that addresses these aspects for time-triggered systems is the *time-triggered model*. The essential feature of this model, as opposed to an asychronous model, is that while the local clocks of individual nodes may not be perfectly synchronized, their disharmony is bounded.

As demonstrated by Rushby, a subset of time-triggered protocols can be systematically shown to implement their synchronous specifications, provided certain timing constraints are met by the underlying system [1]. However, Rushby's work suffers two shortcomings. First, despite his formal verification of the time-triggered model in the PVS mechanical theorem-prover [9], three of the four system assumptions (or formally, axioms) he postulates not only fail to model the actual behavior of time-triggered systems, but are in fact inconsistent. In a recent note by this author, these axioms were mended and their consistency proved [10]; we use the mended axioms herein. Second, the model is too constrained to model some actual realizations of time-triggered protocols. Therefore, we generalize the theory to accommodate time-triggered protocol realizations (and their optimizations) that fall outside the theory developed. Specifically, the theory is extended to accommodate (1) event-triggered behavior, (2) communication delays, (3) reception windows, (4) non-static clock skew, and (5) pipelined rounds (these generalizations are explained and justified in Section 3). These extensions are used to model, for example, the time-triggered protocol implementations of NASA Langley's SPIDER, an ultra fault-tolerant fly-by-wire communications bus [11]. This generalized time-triggered model, a proof of its consistency, and a proof that an arbitrary time-triggered protocol implements its synchronous model have been formulated in PVS (based on Rushby's original PVS specifications and proofs), and those specifications and proofs are available online.[1]

In the second half of this paper, we demonstrate how to formally verify that a protocol realized in hardware satisfies the scheduling constraints of the time-triggered model. The verification is done using SRI's SAL family of model-checkers, which combines a bounded model-checker implementing k-induction with SRI's satisfiability modulo theories (SMT) solver, Yices, to prove LTL safety properties over infinite-state systems (our proofs essentially depend on the theory of linear arithmetic and uninterpreted functions) [12]. These specifications and proofs can also be found on-line.[1] Besides being a novel application of formal verification in general, the approach showcases a particularly successful application of recently-developed infinite-state model-checking techniques.

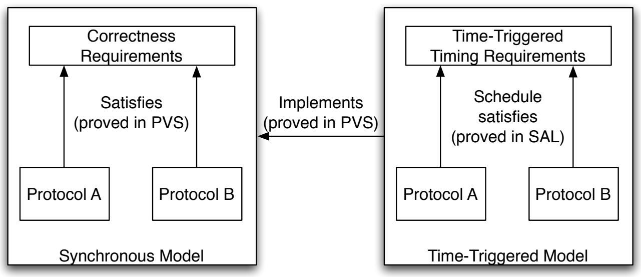
Correctness Requirements
Satisfies (proved in PVS)
Protocol A
Protocol B
Synchronous Model
Implements (proved in PVS)
Time-Triggered Timing Requirements
Schedule satisfies (proved in SAL)
Protocol A
Protocol B
Time-Triggered Model

Figure 1. Time-Triggered Protocol Verification Strategy

Taken together, we have a methodology for proving that a hardware realization of a time-triggered protocol implements its synchronous specification, with respect to its timing parameters, as illustrated in Figure 1.

The remainder of this paper proceeds as follows. The synchronous model's syntax and semantics is provided in Section 2. In Section 3, the syntax and semantics of a generalized time-triggered model are given, and a simulation theorem is stated. In Section 4, we demonstrate how to prove the schedule for a protocol realization satisfies the scheduling constraints of the time-triggered model using as a case-study the SPIDER Distributed Diagnosis Protocol (proofs for the SPIDER Clock Synchronization Protocol and schedule optimizations are provided on-line[1]). Concluding remarks are given in Section 5.

[1]Specification and proof files can be found at `http://www.cs.indiana.edu/~lepike/pub_pages/fmcad.html`. To improve the presentation in this paper, slight syntactic modifications are made from the PVS specifications (e.g., some functions are uncurried).

2 The Synchronous Model

The synchronous model presented is a variant of Lynch's formulation [13] subsequently adapted by Rushby for the purposes of formulating it in a mechanical theorem-prover [1]. Here, we make some slight modifications to the language and also introduce a round-independence relation, to be described shortly. In the synchronous model, distributed protocols are specified as if the nodes in the distributed system execute in lock-step. A synchronous protocol proceeds in rounds. In a round, nodes synchronously and instantaneously update their outbound channels (in the *communication phase*) and then their local state (in the *computation phase*), based on the incoming messages received on their inbound channels [13].

2.1 Syntax

We begin by fixing a set of messages, $mess$. A distinguished element $null$ represents the absence of a message (it can also represent a "do not care" message). Let P be a nonempty set of node identifiers. For each $p \in P$, the following sets and total functions are defined:

- A set of node identifiers, out_nbrs_p, identifying the *outbound neighbors*; i.e., the nodes to which p is connected by outbound channels. A set of node identifiers, in_nbrs_p, identifying the *inbound neighbors*, can be defined from the outbound neighbors:

$$in\_nbrs_p \overset{\text{df}}{=} \{q \in P \mid p \in out\_nbrs_q\}$$

- A set of states, $states_p$. A distinguished component of the state, r, keeps track of the current round. The state $init\_s_p$ is the initial state.

- A *message-generation function* $msg_p : states_p \times out\_nbrs_p \rightarrow mess$ that returns the message p sends to each node to which it is connected by an outbound channel; $null$ is returned if no message is sent.

- A higher-order *state-transition function* $trans_p : states_p \times (in\_nbrs_p \rightarrow mess) \rightarrow states_p$ that returns the new state of p based on the current state and inputs generated by its inbound neighbors.

Sometimes we omit the node-identifier subscript from a set or function to denote a global representation of the system. For example, we define the *global state* to be the function $states \overset{\text{df}}{=} \lambda p.\ states_p$.

Finally, we introduce a *round-independence* relation $independent$ over rounds that holds if messages to be sent in $r+1$ do not depend on the computation that occurs during round r. This relation is used to determine whether a messages for the subsequent round can be sent before computation in the current round is complete. We call this *round-based pipelining*.

2.2 Semantics

The semantics of a synchronous specification can be given by a transition system expressed as a recursive function. The communication phase is modeled by each node applying its msg function, and the computation phase is modeled by each node applying its $trans$ function. The function run takes the number of rounds of execution and the global initial state and returns the final global state (run_p is p's component of the global state returned by run). Thus, for the initial round $init\_rnd$ and the initial state $init\_s$ of a protocol, its behavior can be defined as $run(init\_rnd,\ init\_s)$, where

$$\begin{aligned}
&run(r,\ s) \overset{\text{df}}{=} \\
&\quad \text{if } r = 0 \text{ then } s \\
&\quad \text{else } \lambda p.\ trans_p(run_p(r-1,\ s), \\
&\qquad\qquad\qquad \lambda q.\ msg_q(run_q(r-1,\ s),\ p)), \\
&\text{where } q \in in\_nbrs_p
\end{aligned}$$

The protocols we model execute for only a finite number of rounds. However, a protocol may be scheduled to execute an infinite number of times.

The meaning of the round-independence relation is captured by Axiom 1, which describes the behavior of pipelined communication and computation phases by stating that if the relation holds at round r, then the messages generated from the states in rounds r and $r-1$ are equivalent. The intuition is that if the computation phase of r does not depend on the messages sent in the communication phase of r, then the computation phase may begin before the computation phase ends. We motivate the use of the relation in pipelining optimizations in Section 3.

Semantic Axiom 1 (Pipelining)

$$\begin{aligned}
&\quad \neg independent(0) \\
&\text{and } (\qquad independent(r) \\
&\qquad \text{implies } (\forall q \in out\_nbrs_p : \\
&\qquad\qquad msg_p(run_p(r,\ init\_s),\ q) \\
&\qquad\qquad = msg_p(run_p(r-1,\ init\_s),\ q)))
\end{aligned}$$

3 The Generalized Time-Triggered Model

We extend the synchronous model presented in the previous section to take into account the real-time behavior of the protocols' execution. Some of the syntax comes directly from the original model Rushby developed [1]; the syntactic extensions we introduce for the time-triggered model are noted specifically. After the extensions, we describe the semantics of the model and then present a simulation theorem between an arbitrary protocol in the synchronous and time-triggered models.

Here, we take a moment to motivate informally the generalizations to Rushby's original time-triggered model. We use these generalizations to model the NASA SPIDER protocols and their realizations. We do not know to what extent these generalizations support current realizations of similar time-triggered systems, such as SAFEbus, TTA, and FlexRay [4], but the generalizations can be used to explore more aggressive timing characteristics for any time-triggered system satisfying the model.

Recall the generalizations for which we make provisions: (1) event-triggered behavior, (2) communication delays, (3) reception windows, (4) non-static clock skew, and (5) pipelined rounds; we motivate them in the same order.

Event-Triggered Behavior Some protocols, while mostly time-triggered, occasionally manifest *event-triggered* behavior—actions driven by the observance of some event rather than reaching some pre-scheduled clock-time. A typical example is a clock synchronization protocol such as Davies and Wakerly's protocol [14] or Srikanth and Toueg's protocol [15]. Some of the messages sent in the protocols may be determined by the global schedule, but others are event-triggered: when a node receives some number of messages over its inbound channels, it sends a synchronization (or *echo*) message.

Communication Delays Communication is not instantaneous; latency depends on both the distance a message travels and the medium through which it travels. Latency must be accounted for in tightly synchronized systems with large differences in latency between nodes. Furthermore, for a given distance and medium, there is a nominal latency, and error bounds are also introduced to bound the greatest deviation from the nominal latency that is not regarded as a faulty communication.

Reception Windows Based on the anticipated send time, the expected latency, and the local clock-time, a receiving node will open a *reception window*, which is the set of clock ticks during which the node allows incoming messages in a given round. Messages received outside the window are marked as being faulty. We introduce reception windows into the model to ensure that the windows in a realization do not violate the synchrony assumption.

Non-Static Clock Skew Provisions for reasoning about non-static clock skew have two benefits. First, they allow protocols that satisfy the assumptions of the time-triggered model but nevertheless directly affect the system's timing characteristics (e.g. clock synchronization, self-stabilization, and startup protocols [11]) to be specified in a time-triggered model rather than a more general asynchronous model. Second, they allow for formal reasoning about schedule optimizations. Time-triggered system schedules (also known as *task-descriptor lists* [5]) are usually designed with respect to the maximum possible clock skew during the normal operation of the system. When clocks are not resynchronized, the maximum possible clock skew increases as a linear function of time. If the difference between the possible clock skew at different points in the system's execution is significant, then a schedule can be tightened at those points that the clock skew is small.

Pipelining Rounds Embedded control systems often have hard real-time deadlines that may require aggressive schedules. It may be possible to pipeline the communication and computation rounds of a single protocol or of multiple protocols for better throughput; we call pipelining of this sort *round-based pipelining*. For rounds of the schedule satisfying Axiom 1, the computation phase can begin before the communication phase has completed. Section 4.0.6 mentions how to exploit this in schedules that interleave distinct protocols.

3.1 Syntax

We define *real-time* to be the set of real numbers $\mathbb{R}$ and *clock-time* to be the set of integers $\mathbb{Z}$. Real-time is measured in some arbitrary unit of time (e.g. milliseconds), and clock-time is measured in ticks. By convention, real-time variables and constants are lower-case and clock-time variables and constants are upper-case.

A time-triggered specification extends the syntax for a synchronous specification as follows. (The syntax deals with both the protocol specification and its time-triggered implementation.)

Let P be a nonempty set of node identifiers. For each $p \in P$, the following total functions are defined:

- An *inverse clock* function $C_p : \mathbb{R} \rightarrow \mathbb{Z}$ that takes a real-time as an input and returns a clock-time.

- A *schedule* function $sched_p : \mathbb{N} \rightarrow \mathbb{Z}$ from rounds to clock-times. It parameterizes the communication and computation phase schedules, defined as offsets from the beginning of the round. To accommodate event-triggered behavior, we take a more general view of the schedule function than Rushby does: the schedule function may *determine* the time at which some event occurs, for a time-triggered action, or it may simply denote the clock-time at which an event occurs, for an event-triggered action.

- A relation $sent_p \subseteq out\_nbrs_p \times mess \times \mathbb{R}$, the tuples of which consist of a node q (that is an outbound neighbor of p), a message m, and a real-time t and holds if p sent message m to q at real-time t.

- A relation $recv_p \subseteq in\_nbrs_p \times mess \times \mathbb{R}$, the tuples of which consist of a node q (that is an inbound neighbor of p), a message m, and a real-time t and holds if p received message m from q at real-time t.

In addition, the following functions and constants, not parameterized by node identifiers, are also defined:

- A *schedule discrepancy* function $\Lambda : \mathbb{N} \rightarrow \mathbb{Z}$ from rounds to clock-times denoting the maximum clock-time discrepancy between the schedule functions for that round. This function is added to Rushby's model since nodes may not share the same schedule (due to event-triggered behavior).

- A *communication offset* function $D : \mathbb{N} \rightarrow \mathbb{Z}$ from rounds to a clock-time offset. It determines the clock-time at which nodes send messages in each round.

- A *communication delay* $\delta_{nom} > 0$ is a real-time constant that denotes the expected nominal delay between when a message is sent and when it is received. The small real-time constants $e_l > 0$ and $e_u > 0$ denote the maximum offsets from δ_{nom} at which a message is received sooner ($\delta_{nom} - e_l$) or later ($\delta_{nom} + eu$) than expected, respectively. We require $e_l < \delta_{nom}$ and $e_u < \delta_{nom}$. These constants, added to Rushby's model, provide finer-grained reasoning on the real-time bounds of latency-sensitive time-triggered protocols (e.g., clock synchronization protocols).

- A *computation offset* function $P : \mathbb{N} \rightarrow \mathbb{Z}$ from rounds to a clock-time offset. It determines the clock-time at which nodes begin computation in each round.

- A *maximum drift rate* $\rho \in \mathbb{R}$ such that $0 < \rho < 1$. This is the maximum rate at which a clock may drift.

- A dynamic *clock skew* $\Sigma(r) \geq 0$ function is introduced to Rushby's model, which denotes the greatest clock-time skew occurring between a sender and receiver during the duration of round r.

- A *reception window* function $R : \mathbb{N} \rightarrow \mathbb{Z}$ from rounds to a *reception window offset* is also introduced to Rushby's model. It marks the clock-time at which a node accepts inbound messages. In round r, the reception window closes at $P(r)$.

3.2 System Assumptions and Schedule Constraints

We constrain the interpretations that can be given to the syntax when defining a time-triggered system with the following *system assumptions* and *schedule constraints*. The system assumptions describe the assumed behavior of the underlying system—most notably, the behavior of the local clocks. The schedule constraints ensure the schedule of time-triggered events, given the system assumptions, gives rise to synchronous behavior.

3.2.1 System Assumptions

As in Rushby's original model, we present four system assumptions [1]; recalling that three of Rushby's formulations were inconsistent [10], we present mended and generalized assumptions here. As usual, free variables are implicitly universally-quantified.

Assumption 1 bounds the maximum drift of a clock in terms of the maximum drift rate, ρ.

System Assumption 1 (Clock Drift Rate) *Let* $t_1 \geq t_2$. *Then* $\lfloor(1-\rho)\cdot(t_1-t_2)\rfloor \leq C_p(t_1) - C_p(t_2) \leq \lceil(1+\rho)\cdot(t_1-t_2)\rceil$.

Lemma 1 shows that the clocks are monotonic.

Lemma 1 $t_1 < t_2$ *implies* $C_p(t_1) \leq C_p(t_2)$. *proof By System Assumption 1,* $C_p(t_2) \geq C_p(t_1) + \lfloor(1-\rho)(t_2-t_1)\rfloor$.

Assumption 2 ensures the skew between clocks is no greater than the maximum clock skew so that if any clock is in the communication phase of round r, then all of the clocks are synchronized within the skew of that round. Clock-time $sched_p(r) + D(r)$ is the clock-time at which p sends its messages in round r, and $sched_p(r) + P(r)$ is the clock-time at which it begins the computation phase of round r.

System Assumption 2 (Clock Synchronization)

$$\begin{array}{rl} (& \max(C_p(t),\, C_q(t)) \\ & \geq \min(sched_p(r),\, sched_q(r)) + D(r) \\ \text{and} & \min(C_p(t),\, C_q(t)) \\ & \leq \max(sched_p(r),\, sched_q(r)) + P(r)) \\ \text{implies} & |C_q(t) - C_p(t)| \leq \Sigma(r) \end{array}$$

Assumption 3 ensures that messages are received within the communication delay of when they are sent, modulo error, and that messages received were not "spontaneously generated."

System Assumption 3 (Maximum Communication Delay) *There exists some real-time d, where* $\delta_{nom} - e_l \leq d \leq \delta_{nom} + e_u$*, such that* $sent_p(q, m, t)$ *if and only if* $recv_q(p, m, t+d)$*, and there exists some real-time* d'*, where* $\delta_{nom} - e_l \leq d' \leq \delta_{nom} + e_u$*, such that* $recv_q(p, m, t)$ *if and only if* $sent_p(q, m, t-d')$.

Assumption 4 constrains the maximum discrepancy permitted between the schedule functions of two nodes for a given round. For a particular implementation, whether this constraint is met depends on the constraints for the event-triggered behavior of the individual nodes.

System Assumption 4 $0 \leq |sched_p(r) - sched_q(r)| \leq \Lambda(r)$.

3.2.2 Schedule Constraints

We present six schedule constraints to ensure the time-triggered schedule implements a synchronous system. Constraints 1 - 3 generalize Rushby's original constraints [1], and constraints 4 - 6 are new constraints necessary to constrain pipelining and the scheduling of receivers' reception windows. The schedule constraints are what we later prove hold of the time-triggered schedules in Section 4.

Constraint 1 ensures that the computation offset of round r falls within round r.

Schedule Constraint 1 (Offset Constraint) $0 < P(r) < sched(r+1) - sched(r)$.

Schedule constraint 2 gives the minimum communication offset. Note that if the nominal delay is substantially larger than the clock skew (as is the case in tightly-synchronized systems), the skew has little bearing on when messages can be sent.

Schedule Constraint 2 (Communication Constraint) $D(r) \geq \Sigma(r) + \Lambda(r) - \lfloor(1-\rho)\cdot(\delta_{nom} - e_l)\rfloor$.

Similarly, constraint 3 gives the minimum computation offset. The offset must be greater than the latest time at which a non-faulty message may arive, which is the sum of the communication offset, the clock skew, the maximum schedule discrepancy, and the maximum delay.

Schedule Constraint 3 (Computation Offset Constraint) $P(r) > D(r) + \Sigma(r) + \Lambda(r) + \lceil(1+\rho)\cdot(\delta_{nom} + e_u)\rceil$.

Constraint 4 ensures that pipelining only occurs when the messages to be sent do not depend on the computations from the previous round, and constraint 5 restricts pipelining to consecutive rounds. The effect of pipelining is illustrated in Figure 2.

Schedule Constraint 4 $\neg independent(r)$ *implies* $D(r) \geq 0$.

Schedule Constraint 5 $r > 0$ *implies* $D(r) \geq P(r-1) - sched(r) + sched(r-1)$.

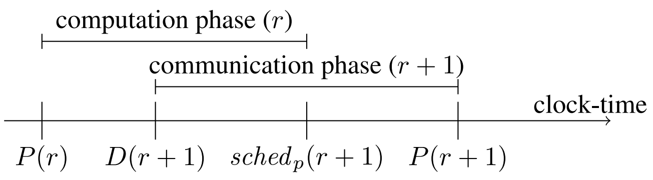

Figure 2. Pipelined Communication Phase (Constraint 5)

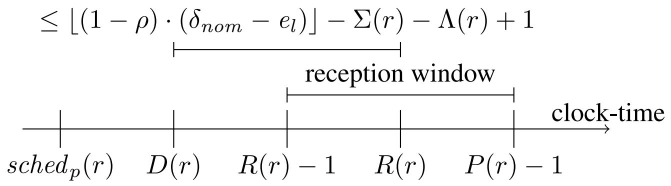
$\leq \lfloor(1-\rho)\cdot(\delta_{nom}-e_l)\rfloor - \Sigma(r) - \Lambda(r) + 1$
reception window
clock-time
$sched_p(r)$
$D(r)$
$R(r)-1$
$R(r)$
$P(r)-1$

Figure 3. Reception Window (Schedule Constraint 6)

The final schedule constraint, Constraint 6, restricts when the reception window is opened. The constraint is illustrated in Figure 3. The reception window must be opened soon enough so that non-faulty messages are received within the window. The formula $\lfloor(1-\rho)\cdot(\delta_{nom}-e_l)\rfloor$ gives a lower bound on the minimum message delay. We add $D(r)$ to take into account the clock-time offset at which the message is sent. The skew for the round, $\Sigma(r)$, is subtracted to account for the case where the receiver's clock is maximally faster than the sender's. A one-tick constant is added to the upper bound on $R(r)$ because the reception window is opened on a clock edge, but messages arrive asynchronously. A message that arrives strictly less than one clock tick before the reception window is opened will be latched on the clock edge when the window is opened.

Schedule Constraint 6 (Reception Window Constraint) $0 \leq R(r) \leq D(r)+\lfloor(1-\rho)\cdot(\delta_{nom}-e_l)\rfloor-\Sigma(r)-\Lambda(r)+1$.

3.3 Semantics

The semantics for the time-triggered model is a transition system in which states are pairs of the form $\langle s, t\rangle$, where s is a global state of the system together with the current real-time, t. The transitions between states are constrained by the axioms given in this section as well as Axiom 1, from the synchronous model.

The axioms are defined over the following uninterpreted functions. We give the type signatures of the functions, as well as their intended interpretations:

- A function $sendtime_p : \mathbb{N} \rightarrow \mathbb{R}$ from rounds to real-times denoting the real-time that p broadcasts messages in each round.
- A *time-triggered system state* function $ttss_p : states \times \mathbb{Z} \rightarrow states_p$ that takes a global state s, a clock-time T, and returns p's state after executing for T clock ticks from s.
- A *time-triggered inbound messages* function $ttin_p : \mathbb{Z} \times in\_nbrs_p \rightarrow mess$ that maps a clock-time T and an inbound neighbor q to the message p receives from q at T.
- A *time-slice* function $gs : \mathbb{N} \rightarrow \mathbb{R}$ from rounds to real-times. Its purpose is to provide real-times at which the system state of the time-triggered model of a protocol is the same as the untimed model of the protocol, for each round.

Axiom 2 constrains the $sendtime_p$ function by ensuring that at the real-time that p broadcasts its message in round r, its clock-time is at the communication offset into that round.

Semantic Axiom 2 $C_p(sendtime_p(r)) = sched_p(r) + D(r)$.

Axioms 3 and 4 constrain the behavior of the $sent_p$ function by first stating the sufficient conditions for it to hold and then the necessary conditions for it to hold. Axiom 3 ensures that the message p sends to q at the real-time $sendtime_p(r)$ is the message generated by its message-generation function using its time-triggered state at the beginning of round r. Axiom 4 ensures that if the $sendtime_p$ relation is satisfied, then it is satisfied by a message generated by the message-generation function in some round and by the real-time at the communication delay into the round.

Semantic Axiom 3 $sent_p(q,\, msg_p(ttss_p(s,\, sched_p(r) + D(r)),\, q),\, sendtime_p(r))$*, where* $q \in out\_nbrs_p$.

Semantic Axiom 4 $sent_p(q,\, m,\, t)$ *implies there exists a round* r *such that* $t = sendtime_p(r)$ *and* $m = msg_p(ttss_p(s,\, sched_p(r) + D(r)),\, q)$*, where* $q \in out\_nbrs_p$.

Before stating the next axiom, we define the relation $recv\_win\_open_p$, which takes a real-time t and a round r and is true if the real-time falls within p's reception window for round r. Messages may arrive strictly less than one clock tick before $R(r)$ is reached, but these messags are latched at $R(r)$. Therefore, $recv\_win\_open_p(t,\, r)$ holds for any real-time t that is mapped to a clock-time strictly greater than $R_p(r) - 1$ (and strictly less than the beginning of the computation phase).

Definition 1 (*Reception Window Open*)

$$recv\_win\_open_p(t,\, r) \overset{\text{df}}{=} \\ sched_p(r) + R_p(r) - 1 \le C_p(t) < sched_p(r) + P(r)$$

Axiom 5 constrains the behavior of the $ttin_p$ function by ensuring that for any clock-time T in the computation phase of round r, $ttin_p(T,\, q)$ is the message p receives from q in the reception window of round r (ϵ is Hilbert's choice operator).

Semantic Axiom 5 $sched_p(r) + P(r) \le T < sched_p(r+1)$ *implies* $ttin_p(T,\, q) =$

$$\epsilon\left(\left\{\begin{array}{ll} m \in mess \mid \exists t \in \mathbb{R}. & recv\_win\_open_p(t,\, r) \\ & \text{and } recv_p(q,\, m,\, t) \end{array}\right\}\right)$$

Axioms 6 and 7 constrain the $ttss_p$ function. Axiom 6 determines p's time-triggered state at the clock-time $sched(r)$, for each round r, to be the current state if $r = 0$, or the state computed in the computation phase of the previous round.

Semantic Axiom 6

$$\begin{array}{l} ttss_p(s,\, sched_p(r)) = \\ \quad \text{if } r = 0 \text{ then } s_p \\ \quad \text{else } trans_p(ttss_p(s,\, T),\, \lambda q.\ ttin_p(T,\, q)) \\ \text{where } q \in in\_nbrs_p \text{ and} \\ \qquad T = sched_p(r-1) + P(r-1) \end{array}$$

Axiom 7 ensures that outside of the computation phase, p's time-triggered state does not spontaneously change.

Semantic Axiom 7 *For all clock-times* T, $sched_p(r) \le T \le sched_p(r) + P(r)$ *implies* $ttss_p(s,\, T) = ttss_p(s,\, sched_p(r))$.

Finally, Axiom 8 constrains the real-time $gs(r)$ to be the real-time at which the process with the slowest clock has reached $sched(r)$.

Semantic Axiom 8 *For all nodes* l,

$$\begin{array}{r} \forall q:\ C_q(gs(r)) \ge sched_l(r)) \\ \text{and } \exists p:\ C_p(gs(r)) = sched_l(r)) \end{array}$$

Finally, Axiom 9 ensures that while a node is in its computation phase, its state is either the state it has before applying its state-transition function in that round or the updated state resulting from its application (in this model, the state is updated at some nondeterministic time during the computation phase, but the entire state is updated instantaneously).

Semantic Axiom 9 *For all clock-times* T, $sched_p(r) + P(r) \le T < sched_p(r+1)$ *implies either* $ttss_p(s,\, T) = ttss_p(s,\, sched_p(r))$, *or* $ttss(s,\, T) = ttss(s,\, sched_p(r+1))$.

An interpreted transition relation is one that satisfies axioms 1 through 9. The axiomatization ensures a simulation relation exists between a synchronous protocol and its time-triggered implementation, as stated in Theorem 1.

Theorem 1 $ttss_p(s,\, C_p(gs(r))) = run_p(r,\, init\_s)$. *proof By induction on the rounds of the protocol; see [16] for a proof sketch of the proof formulated in PVS.*[1]

4 Schedule Verification

The schedule of a time-triggered protocol's realization are the clock-times at which events are scheduled to occur. Assuming that an architecture is fixed and satisfies the system assumptions, we wish to prove the schedule developed for a protocol's realization satisfies the six schedule constraints, Constraint 1 through Constraint 6, from Section 3.2.2.

This verification is carried out in SRI's SAL family of model-checkers, which contains an infinite-state bounded model checker that combines the Yices SMT solver with the k-induction model-checking algorithm to make bounded model-checking complete for safety properties [12]. Because the languages of PVS and SAL are similar, the schedule constraints have nearly identical formulations in the respective languages.

The verification technique is demonstrated by verifying the schedule constraints for two SPIDER time-triggered protocols. The schedules verified are taken from the VHDL coded by Wilfredo Torres-Pomales and Mahyar Malekpour of the NASA Langley Research Center, the implementors of the latest prototype [11]. The schedules were generated using Matlab® according to the by-hand analysis of the timing requirements [11]. The verification technique provides a formal mapping from the synchronous specification of these protocols to the time-triggered implementation. Below, we overview the verification of the SPIDER Distributed Diagnosis Protocol (SPIDER DD Protocol) schedule. The verification of the constraints for a more complex protocol, the SPIDER Clock-Synchronization Protocol, as well as a demonstration of how to use this technique to optimize the throughput of SPIDER protocols, are available on-line.[1]

Briefly, the SPIDER DD Protocol ensures nodes maintain a consistent assignment of the faultiness of the other nodes [11]. Nodes may individually accuse one another of being faulty, based on accumulated evidence. During the protocol, if enough nodes accuse a node, the accusations are promoted to an agreed-upon *conviction*. When a node has been convicted, the other non-faulty nodes ignore the convicted node until it proves itself to be non-faulty. (The mechanism for doing so involves executing the SPIDER Reintegration Protocol [17].)

The verification of this protocol's schedule is straightforward. The protocol has four rounds. The schedule offsets

D, P, and R do not vary from round to round. We verify the protocol with respect to the maximum possible skew for the duration of the protocol. Furthermore, none of the rounds are pipelined, and there are no event-triggered actions.

4.0.1 Type and Constant Declarations

The type and constant declarations are straightforward in SAL. All system constants are interpreted to be concrete values taken from the system parameters for the targeted prototype. The SAL specification of the declarations are given in Figure 4. The schedule constraints require taking the floor and ceiling of the minimum and maximum communication delay, respectively; we do this by hand (The Yices SMT solver cannot handle non-linear arithmetic) and set them equal to constants.

```
REALTIME : TYPE = REAL;
CLOCKTIME : TYPE = INTEGER;
OFFSET : TYPE = {T: CLOCKTIME | T >= 0};
RND : TYPE = NATURAL;
rho : REALTIME = 1/10000;
d_nom : {t: REALTIME | t >= 0} = 5;
ERROR : TYPE = {t: REALTIME |
  t >= 0 AND t < d_nom};
e_l : ERROR = 5/10000;
e_u : ERROR = 5/10000;
% floor((1 - rho) * (d_nom - e_l))
fl_d_min : CLOCKTIME = 4;
% ceiling((1 + rho) * (d_nom + e_u))
cd_d_max : CLOCKTIME = 6;
```

Figure 4. Type and Constant Declarations

4.0.2 Variables

In SAL, we build a state-machine to model-check. The state-machine transitions follow the order of the schedule's rounds and update state variables accordingly. Therefore, in the SAL model, we replace some of the mathematical functions from the time-triggered system model presented in Section 3 with corresponding state variables ranging over rounds. Thus, the set of state variables include $sched$, D, P, R, Λ, Σ, $independent$, and R. The state variables may be nondeterministically updated in the state-machine transitions depending on the specifics of the schedule being verified. In the schedule verified for the SPIDER DD Protocol, the values of D, P, R, and Σ are constant over the rounds for this protocol's schedule; only $sched$ is updated from round to round. The other protocol schedule verifications are more complex; see 4.0.6.

4.0.3 Schedule Constraint Specification

The schedule constraints stated in Section 3.2.2 are stated in SAL as shown in Figure 5. Some of these constraints compare the schedule between successive rounds (e.g., Constraint 1). Because we have transcribed the functions over the rounds to variables that are updated in the state machine at each round, these relations may take as arguments the values of these variables in a round and compare them to the values in the next round (e.g., `constraint1` takes `pre_sched` and `sched` as arguments, denoting the values for $sched(r-1)$ and $sched(r)$, respectively). The SPIDER DD Protocol contains no event-triggered behaviors; therefore, for all rounds, the schedule skew Λ is zero. We therefore omit it from the constraints.

```
constraint1(P: OFFSET, pre_sched: CLOCKTIME,
            sched: CLOCKTIME): BOOLEAN =
  0 < P AND P < sched - pre_sched;

constraint2(D: CLOCKTIME, S: OFFSET): BOOLEAN =
  D >= S - fl_d_min;

constraint3(P: OFFSET, D: CLOCKTIME, S: OFFSET):
  BOOLEAN = P > D + S + cd_d_max;

constraint4(r: RND, D: CLOCKTIME): BOOLEAN =
  (NOT independent?(r)) => D >= 0;

constraint5(pre_P: OFFSET, D: CLOCKTIME,
            pre_sched: CLOCKTIME, sched: CLOCKTIME):
  BOOLEAN = D >= pre_P - sched + pre_sched;

constraint6(D: CLOCKTIME, R: CLOCKTIME, S: OFFSET):
  BOOLEAN = R - 1 <= D + fl_d_min - S;
```

Figure 5. SAL Specification of the Generalized System Assumptions

4.0.4 Specifying a Round-Based Schedule

We create a state-machine representation of how the schedule constraints evolve through the rounds of execution. In addition to schedule variables, for each constraint, a boolean variable is declared. The value of the variable is determined by whether its associated schedule constraint is satisfied in the present round. The state machine includes a counter `r` that records the current round in the synchronous abstraction of the protocol. In each initial state, this counter is set to 0. Each transition from a state in round r is to a state in round $r+1$. In general, we check the schedule constraints for the next-state values of the variables. For constraints that compare the values between rounds, the current-state variable values and the next-state variable values are compared in the constraint. Because there are no previous state assignments in round 0, those state variables associated with constraints that compare values between rounds are declared to be true upon initialization.

Not every state variable needs to be updated in each transition. If a state variable is not reassigned in a guarded transition, its value remains the same in the next state.

4.0.5 Verification

The property stating that in all reachable states, each constraint holds can then be specified by the following LTL state invariant.

```
constraints: LEMMA SYSTEM |-
  G(c1 AND c2 AND c3 AND c4 AND c5 AND c6);
```

The property is verified by executing SAL's infinite-state bounded model checker. The lemma `constraints` is verified by the k-induction solver, for $k = 2$. The proof is fully automatic and requires no supporting invariants.

4.0.6 Other Verifications

We have also used this proof technique to verify the SPIDER Clock Synchronization Protocol schedule. The purpose of the clock synchronization protocol is to resynchronize the local clocks, in the presence of faults, after they have possibly drifted apart [11]. Consequently, the schedule of the clock synchronization protocol is more complex, as the skew is a function of the round of the protocol.

Similarly, we have also verified a schedule interleaving distinct SPIDER protocols. In these schedules, we can take advantage of the round-based pipelining optimizations. Both of these verifications are available on-line.[1]

5 Discussion

Faults are not dealt with explicitly in the models presented; we discuss them below. Concluding remarks follow.

5.1 Faults

Neither the synchronous nor time-triggered model presented explicitly model faulty behavior. Nevertheless, because many time-triggered protocols are fault-tolerant, we ultimately wish to prove that the protocols behave correctly in the presence of faults. Before discussing faults specifically, recall that the state-transition function $trans$ and the message-generation function msg are left uninterpreted, and the same functions appear in both the synchronous and time-triggered models. Thus, the simulation theorem holds regardless of their instantiations; in particular, the functions can be partially-interpreted and under-specify a protocol, and the theorem still holds.

Keeping this in mind, faults can be modeled, as Rushby notes, by partially-interpreting $trans$ and msg, allowing them to return arbitrary values, nondeterministically, if a node or channel is faulty [1]. In fact, all faulty behavior can be modeled, with no loss of fidelity, by partially-interpreting the message-generation function only [18].

In the synchronous model, the correctness of a protocol can be verified under a *maximum fault assumption* (MFA), which constrains the kinds of faults, and the number of each kind, under which the protocol is hypothesized to behave correctly [1]. If the effects of faults are captured by the message-generation function msg, then the MFA can be thought of as a constraint on the function's nondeterminism. Thus, one purpose of the time-triggered model is to *define* the timing behaviors that are non-faulty. That is, if the timing characteristics of the system satisfy the system assumptions and schedule constraints of the time-triggered model, then no faults will result if these characteristics hold in a realization (timing and value faults may still arise from environment factors).

5.2 Concluding Remarks

The approach presented herein and illustrated in Figure 1 is one portion of an end-to-end verification methodology—from the distributed protocols to the hardware implementations of the nodes—in a time-triggered system. As an anonymous reviewer notes, the results presented herein can be combined with recent work in physical-layer protocol verification [19] and gate-level I/O device verification [20] to further the goal of an end-to-end verification of time-triggered systems.

6 Acknowledgments

This paper is a revised portion of a recent dissertation [16]. We thank Steve Johnson (who advised the dissertation), Paul Miner, Geoffrey Brown, Larry Moss, Wilfredo Torres-Pomales, and our anonymous reviewers for their ideas and comments. Finally, this paper owes a large debt to John Rushby's original work on the topic.

References

[1] J. Rushby, "Systematic formal verification for fault-tolerant time-triggered algorithms," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 651–660, September 1999.

[2] P. Koopman, Ed., *Critical Embedded Automotive Networks*, ser. IEEE Micro, vol. 22–4. IEEE Computer Society, July/August 2002.

[3] P. Traverse, I. Lacaze, and J. Souyris, "Airbus fly-by-wire - a total approach to dependability," in *IFIP Congress Topical Sessions*, 2004, pp. 191–212.

[4] J. Rushby, "Bus architectures for safety-critical embedded systems," in *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*, ser. Lecture Notes in Computer Science, T. Henzinger and C. Kirsch, Eds., vol. 2211. Lake Tahoe, CA: Springer-Verlag, Oct. 2001, pp. 306–323.

[5] H. Kopetz, *Real-Time Systems*. Kluwer Academic Publishers, 1997.

[6] R. W. Butler and G. B. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *Software Engineering*, vol. 19, no. 1, pp. 3–12, 1993, available at http://citeseer.nj.nec.com/butler93infeasibility.html.

[7] P. Miner, A. Geser, L. Pike, and J. Maddalon, "A unified fault-tolerance protocol," in *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*, ser. LNCS, Y. Lakhnech and S. Yovine, Eds., vol. 3253. Springer, 2004, pp. 167–182, available at http://www.cs.indiana.edu/~lepike/pub_pages/unified.html.

[8] H. Pfeifer, "Formal analysis of fault-tolerant algorithms in the time-triggered architecture," Ph.D. dissertation, Universität Ulm, 2003, available at http://www.informatik.uni-ulm.de/ki/Papers/pfeifer-phd.html.

[9] S. Owre, J. Rusby, N. Shankar, and F. von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 107–125, February 1995.

[10] L. Pike, "A note on inconsistent axioms in rushby's "systematic formal verification for fault-tolerant time-triggered algorithms"," *IEEE Transactions on Software Engineering*, vol. 32, no. 5, pp. 347–348, May 2006, available at http://www.cs.indiana.edu/~lepike/pub_pages/time_triggered.html.

[11] W. Torres-Pomales, M. R. Malekpour, and P. Miner, "ROBUS-2: A fault-tolerant broadcast communication system," NASA Langley Research Center, Tech. Rep. NASA/TM-2005-213540, 2005.

[12] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Computer-Aided Verification, CAV 2004*, ser. Lecture Notes in Computer Science, R. Alur and D. Peled, Eds., vol. 3114. Boston, MA: Springer-Verlag, July 2004, pp. 496–500.

[13] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[14] D. Davies and J. F. Wakerly, "Synchronization and matching in redundant systems," *IEEE Transactions on Computers*, vol. 27, no. 6, pp. 531–539, June 1978.

[15] T. K. Srikanth and S. Toueg, "Optimal clock synchronization," *Journal of the ACM*, vol. 34, no. 3, pp. 626–645, July 1987.

[16] L. Pike, "Formal verification of time-triggered systems," Ph.D. dissertation, Indiana University, 2006, available at http://www.cs.indiana.edu/~lepike/phd.html.

[17] L. Pike and S. D. Johnson, "The formal verification of a reintegration protocol," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA: ACM Press, 2005, pp. 286–289, available at http://www.cs.indiana.edu/~lepike/pub_pages/emsoft.html.

[18] L. Pike, J. Maddalon, P. Miner, and A. Geser, "Abstractions for fault-tolerant distributed system verification," in *Theorem Proving in Higher Order Logics (TPHOLs)*, ser. LNCS, K. Slind, A. Bunker, and G. Gopalakrishnan, Eds., vol. 3223. Springer, 2004, pp. 257–270, available at http://www.cs.indiana.edu/~lepike/pub_pages/abstractions.html.

[19] G. M. Brown and L. Pike, "Easy parameterized verification of biphase mark and 8N1 protocols," in *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, 2006, pp. 58–72, available at http://www.cs.indiana.edu/~lepike/pub_pages/bmp.html.

[20] S. Knapp and W. Paul, "Realistic worst case execution time analysis in the context of pervasive system verification," in *Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*, ser. LNCS volume 4444. Springer, 2006, pp. 53–81.